View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Juelich Shared Electronic Resources

John von Neumann Institute for Computing

Parallelization of Multilevel Preconditioners Constructed from Inverse-Based ILUs on Shared-Memory Multiprocessors

José I. Aliaga, Matthias Bollhöfer, Alberto F. Martín, Enrique S. Quintana-Ortí

published in

Parallel Computing: Architectures, Algorithms and Applications ,
C. Bischof, M. Bücker, P. Gibbon, G.R. Joubert, T. Lippert, B. Mohr, F. Peters (Eds.),
John von Neumann Institute for Computing, Jülich,
NIC Series, Vol. **38**, ISBN 978-3-9810843-4-4, pp. 287-294, 2007.
Reprinted in: Advances in Parallel Computing, Volume **15**,
ISSN 0927-5452, ISBN 978-1-58603-796-3 (IOS Press), 2008.

© 2007 by John von Neumann Institute for Computing
Permission to make digital or hard copies of portions of this work for personal or classroom use is granted provided that the copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise requires prior specific permission by the publisher mentioned above.

http://www.fz-juelich.de/nic-series/volume38

Parallelization of Multilevel Preconditioners Constructed from Inverse-Based ILUs on Shared-Memory Multiprocessors

José I. Aliaga[1], **Matthias Bollhöfer**[2], **Alberto F. Martín**[1], **and Enrique S. Quintana-Ortí**[1]

[1] Depto. de Ingeniería y Ciencia de Computadores, Universidad Jaume I, 12.071–Castellón, Spain
E-mail: {*aliaga, martina, quintana*}*@icc.uji.es*

[2] Institute of Computational Mathematics, TU-Braunschweig, D-38106 Braunschweig, Germany
E-mail: m.bollhoefer@tu-braunschweig.de

In this paper, we present an OpenMP parallel preconditioner based on ILUPACK. We employ the METIS library to locate independent tasks which are dynamically scheduled to a pool of threads to attain a better load balance. Experimental results on a shared-memory platform consisting of 16 processors report the performance of our parallel algorithm.

1 Introduction

The solution of linear systems and eigenvalue problems is ubiquitous in chemistry, physics, and engineering applications. When the matrix involved in these problems is large and sparse, iterative methods as, e.g., those based on Krylov subspaces, are traditionally employed in the solution of these problems[1]. Among these methods, ILUPACK[a] (Incomplete LU decomposition PACKage) is a novel software package based on approximate factorizations which enhances the performance of the process in terms of a more accurate solution and a lower execution time.

In order to reduce the time that is needed to compute the preconditioner or the execution time per iteration of a linear system solver, we can use high-performance computing techniques to better exploit the platform where the problem is to be solved. In this paper we pursue the parallelization using OpenMP[2] of the computation of a preconditioner, based on ILUPACK, for the solution of linear systems with symmetric positive definite (s.p.d.) coefficient matrix. The target architecture, shared-memory multiprocessors (SMMs), includes traditional parallel platforms in scientific computing, such as symmetric multiprocessors (SMPs), as well as the novel multicore processors. OpenMP provides a natural, simple, and flexible application programming interface for developing parallel applications for parallel architectures with shared-memory (and we assume that it will continue to do so for future multicore systems).

The paper is structured as follows. In Section 2 we briefly review ILUPACK. Next, in Section 3, we offer some details on the parallel preconditioner. Section 4 then gathers data from our numerical experiments with the parallel algorithm, and a few concluding remarks and future research goals follow in Section 5.

[a] `http://www.math.tu-berlin.de/ilupack`.

2 An Overview over ILUPACK

ILUPACK includes C and Fortran routines to solve linear systems of the form $Ax = b$ via (iterative) Krylov subspace methods: the package can be used to both compute a preconditioner and apply it to the system. We will focus on the computation of the preconditioner since this is the most challenging task from the parallelization viewpoint. The routines that perform this task in ILUPACK sum more than 4000 lines of code. Hereafter we will consider the coefficient matrix A to be s.p.d.

The rationale behind the computation of the preconditioner is to obtain an incomplete LU decomposition of A, while avoiding computations with "numerically-difficult" diagonal pivots, which are moved to the last rows by applying a sequence of permutations, P. To do that, the Crout variant[1] of the LU decomposition is used, so that the following computations are performed in each step of the procedure:

1. Apply the transformations corresponding to the part of the matrix that is already factored to the current row and column of the matrix.
2. If the current pivot is "numerically dubious", move the current row and column to the last positions of the matrix and accumulate this permutation on P.
3. Otherwise, proceed with the factorization of the current row and column of the matrix and apply certain "dropping techniques".

When this process is finished a partial ILU decomposition of $P^T AP$ is obtained, and a Schur complement must be computed for that part of the permuted matrix that was not factored. The process is recursively repeated on the Schur complement until the matrix is fully factored, yielding a *multilevel* ILU decomposition of the permuted matrix. The dropping techniques and the computation of the Schur complement are designed to bound the elements of the inverses of the triangular factors in magnitude. This property improves the numerical performance of the method, as the application of the preconditioner involves those inverses[3–6].

3 An OpenMP Parallel Preconditioner

In this section we describe our parallel preconditioner. It is important to realize that this is not a parallel implementation of the serial preconditioner in ILUPACK, but a parallel algorithm for the computation of a preconditioner that employs the serial routines in ILUPACK. The computations (stages and operations) as well as the results (preconditioners) obtained by ILUPACK and our parallel algorithm are, in general, different.

The first step in the development of a parallel preconditioner consists in splitting the process into tasks and identifying the dependencies among these. After that, tasks are mapped to threads deploying task pools[7], in an attempt to achieve dynamic load balancing while fulfilling the dependencies.

Frequently, the initial ordering of the sparse coefficient matrix is not suitable to parallel factorization, because it is not possible to identify a number of independent tasks sufficiently large or because the costs of the tasks that would result are highly unbalanced. In these cases, a different ordering (permutation) of the matrix may help to expose more parallelism. In particular, the MLND (Multilevel Nested Dissection) algorithm included

in METIS[8] usually leads to balanced elimination trees which exhibit a high degree of concurrency. Figure 1 illustrates the original sparsity pattern of the *Apache1* matrix from the University of Florida (UF) sparse matrix collection[9], and the pattern of the matrix ordered using MLND.

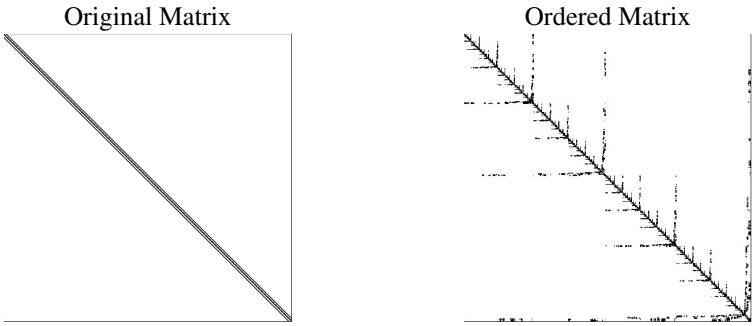

Figure 1. Sparsity pattern of the *Apache1* matrix. Left: original. Right: matrix ordered using MLND.

Tasks for the parallel algorithm and their dependencies can be identified by manipulating the elimination tree[10] of the ordered matrix: If we condense each elimination subtree rooted at height $log_2(p)$, where p is the number of processors (threads), we obtain a set of tasks which is organized in a tree-like structure, with nodes representing tasks, and the ancestor-descendant relationship representing dependencies among them; see Fig. 2. This task tree defines a partition of the ordered matrix which identifies matrix blocks that can be factorized in parallel and matrix blocks whose factorization depends on other computations. Figure 3 represents the partition defined by a task tree of height 1. The factorization of the leading blocks A_{11}, A_{22}, and their corresponding subdiagonal blocks, can be performed in parallel. However, the factorization of A_{33} depends on the results produced by the factorizations of the two leading diagonal blocks.

In order to perform the parallel factorization of the matrix, we assign a submatrix to each leaf of the tree; see the bottom of Fig. 3. There, A^1_{33} and A^2_{33} comply with $A_{33} = A^1_{33} + A^2_{33}$. Driven by the multilevel ILU scheme, the parallel algorithm computes the factorization of the leading diagonal block of each leaf submatrix, and then obtains the corresponding Schur complements. To do this computations, the parallel algorithms employs the serial routines in ILUPACK. Upon completion of this process, the root task appropriately combines the Schur complements of each factorization, creating a submatrix which can be fully factorized. This process is easily generalized for task trees of height larger than 1.

Due to the properties of the MLND ordering, the major part of the computation is concentrated on the leaves of the task tree; therefore a good load balance in the computational costs of the tasks associated with the leaves is mandatory to achieve high parallel performance. Figure 2 shows the "estimated" cost (in terms of the number of nonzero elements of the corresponding submatrix) of the leaves for the MLND-ordered *Apache1* matrix. Although MLND ordering performs a best-effort work, there are a few leaves that concentrate the major part of the computation. In order to attain a better load balance, our parallel al-

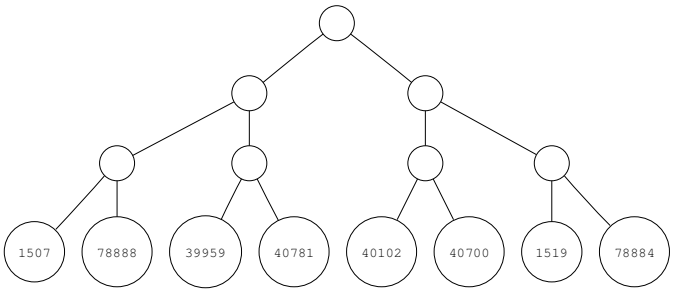

Figure 2. Non-split task tree for the MLND-ordered *Apache1* matrix. The labels in the nodes represent the number of nonzero elements in the corresponding submatrix. These values are used as an estimation of the cost of the associated task.

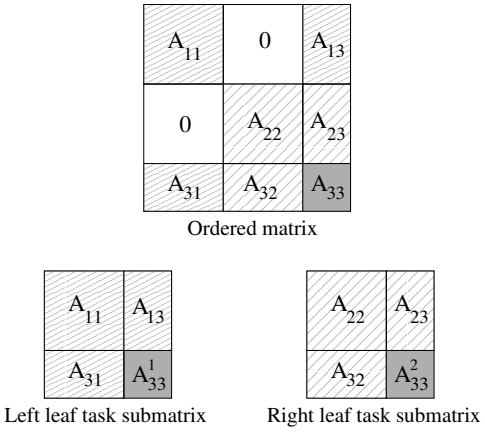

Figure 3. Top: ordered matrix partition defined by an example task tree of height 1. Bottom: submatrices created on leaf tasks.

gorithm splits those tasks with cost higher than a given threshold into finer-grain tasks; see the split task tree for the MLND-ordered *Apache1* matrix in Fig. 4. MLND ordering is a must in our current implementation of the parallel algorithm as it forms the basis for the identification of concurrent tasks; on the other hand, this ordering is optional for the serial algorithm in ILUPACK. Task splitting in the parallel algorithm generally forces a different number of levels compared with those of the serial algorithm; therefore, the stages and operations that are performed by these two algorithms also differ.

The task tree is constructed sequentially, before the (true) computation of the preconditioner commences. All leaf tasks in this tree are initially inserted in the *ready queue* which, at any moment, contains those tasks with all dependencies fulfilled. Tasks are dequeued from the head and enqueued at the tail of this structure. In order to balance the load, the execution of the leaf tasks with higher computational cost is priorized by inserting them first in the queue. The execution of tasks is scheduled dynamically: as threads become idle, they monitor the queue for work (pending tasks). When a thread completes execution

of a task, all tasks dependent on it are examined and those with their dependencies fulfilled are enqueued at the ready queue by this thread. Idle threads continue to dequeue tasks until all tasks have been executed. Similar mechanisms have been proposed for irregular codes as part of the Cilk project[11] and for dense linear algebra codes in the FLAME project[12].

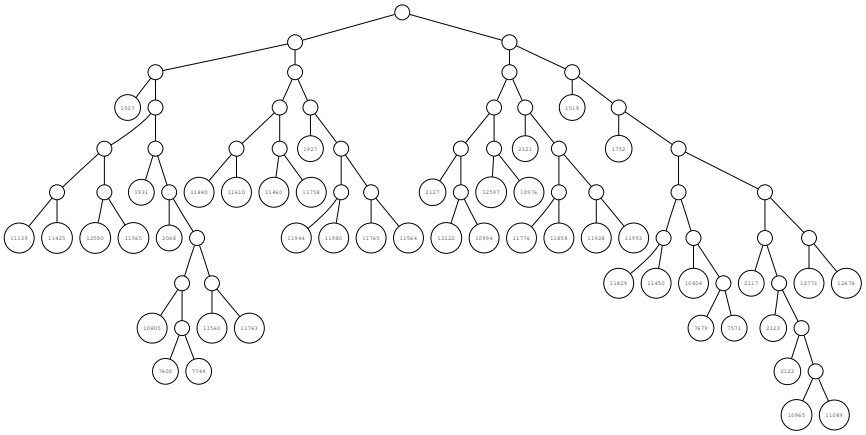

Figure 4. Split task tree for the MLND-ordered *Apache1* matrix. The labels in the nodes represent the number of nonzero elements in the corresponding submatrix. These values are used as an estimation of the cost of the associated task.

4 Experimental Results

All experiments in this section were obtained on a SGI Altix 350 CC-NUMA multiprocessor consisting of 16 Intel Itanium2@1.5 GHz processors sharing 32 GBytes of RAM via a SGI NUMAlink interconnect. No attempt is made to exploit the data locality on this CC-NUMA architecture. IEEE double-precision arithmetic was employed in all the experiments, and one thread was scheduled per processor. In both the serial and parallel algorithms we used ILUPACK default values for the condition estimator (`condest`=100), and the factor and Schur tolerances (`tol1`=10^{-2}, `tol2`=10^{-2}) for the dropping thresholds in the triangular factors and their complements.

Table 1 characterizes the benchmark matrices from the UF sparse matrix collection[9] employed in the evaluation, and reports the results obtained from the execution of the serial algorithm in ILUPACK: serial execution time and fill-in factor (ratio between the number of nonzero elements in the triangular factors produced by the algorithm and the matrix) for the matrix preprocessed using MLND ordering.

In the parallel algorithm, a task is further subdivided into two subtasks when the ratio between the number of nonzero entries of the complete matrix and the submatrix associated with the task was larger than p (option A) or $2p$ (option B), with p the number of processors/threads. Table 2 reports the number of leaves in the task trees that these values produce. Clearly option A yields a smaller number of leaves, and thus a smaller degree of

Code	Matrix name	Rows/Cols.	Nonzeros	Time (s)	Fill-in factor
M1	*GHS_psdef/Apache1*	80800	542184	2.81	4.8
M2	*Schmid/Thermal1*	82654	574458	1.32	4.2
M3	*Schenk_AFE/Af_0_k101*	503625	17550675	24.5	2.7
M4	*GHS_psdef/Inline_1*	503712	36816342	143	4.8
M5	*GHS_psdef/Apache2*	715176	4817870	36.9	5.8
M6	*GHS_psdef/Audikw_1*	943695	77651847	320	4.1

Table 1. Matrices selected to test the parallel multilevel ILU algorithm (left) and results (execution time and fill-in factor) from the execution of the serial algorithm in ILUPACK (right).

	Option A														
Matrix	#Leaf tasks					c_w (%)					Fill-in factor				
M1	2	4	11	23	24	0	0	57	58	58	4.8	4.8	4.8	4.7	4.7
M2	3	5	9	17	19	70	49	35	24	31	4.1	4.1	4.2	4.2	4.2
M3	2	4	9	16	18	0	0	4	4	20	2.7	2.6	2.6	2.5	2.5
M4	3	6	12	16	21	42	30	29	14	32	4.4	4.2	3.8	3.6	3.5
M5	2	7	11	27	27	0	85	60	78	78	5.8	5.8	5.9	6.0	6.0
M6	2	4	8	16	**17**	0	2	4	6	**15**	3.7	3.5	3.4	3.4	3.4
#Procs.	2	4	8	12	16	2	4	8	12	16	2	4	8	12	16

	Option B														
Matrix	#Leaf tasks					c_w (%)					Fill-in factor				
M1	4	11	24	43	45	0	58	58	47	46	4.8	4.8	4.7	4.3	4.2
M2	5	9	19	33	37	50	35	31	18	25	4.1	4.2	4.2	4.1	4.0
M3	4	9	18	32	37	0	5	20	07	19	2.6	2.6	2.5	2.3	2.2
M4	6	12	21	33	42	30	29	32	16	28	4.2	3.8	3.5	3.2	3.1
M5	7	11	27	50	**50**	85	60	78	67	**67**	5.8	5.9	6.0	6.1	6.1
M6	4	8	**17**	32	32	2	4	**15**	7	7	3.5	3.4	3.4	3.4	3.4
#Procs.	2	4	8	12	16	2	4	8	12	16	2	4	8	12	16

Table 2. Number of leaf tasks, coefficient of variation, and fill-in generated by the parallel algorithm using 2, 4, 8, 12, and 16 processors.

parallelism, but the tasks present higher granularity. As a measure of how similar the estimated costs of the leaves are, the table also shows the coefficient of variation $c_w = \sigma_w/\bar{w}$, with σ_w the standard deviation and $\bar{w}$ the average of these costs. A ratio close to 0 (e.g., in Option A/M1/4 processors) indicates that all leaves have very similar costs, while a ratio closer to 100% (e.g., Option A/M5/4 processors) indicates a high variability of the costs that could be the source for an unbalanced distribution of the computational load. Finally, the last column of the table reports the fill-in factor produced by the parallel algorithm, which are similar to those attained by the serial algorithm in ILUPACK. As the number of processors or the number of tasks are increased, the fill-in factor tends to be reduced.

Table 3 reports the execution time and the speed-up of the parallel algorithm. The speed-up is computed with respect to the parallel algorithm executed using the same task

Matrix	Option A Time (secs.)					Speed-up				
M1	1.36	0.66	0.38	0.32	0.23	1.97	3.87	6.37	7.19	10.18
M2	0.64	0.32	0.17	0.16	0.01	1.99	3.81	7.20	7.60	12.53
M3	12.2	6.13	3.24	2.13	1.8	1.96	3.89	7.26	11.09	13.02
M4	64.9	36.7	16.9	15.4	9.28	2.01	3.41	6.69	7.13	11.35
M5	18.2	9.44	7.80	3.40	2.61	1.99	3.84	4.54	10.43	13.61
M6	152	89.4	68.5	51.8	**51.2**	1.88	3.25	5.38	9.69	**9.92**
#Procs.	2	4	8	12	16	2	4	8	12	16

Matrix	Option B Time (secs.)					Speed-up				
M1	1.29	0.64	0.35	0.27	0.18	1.97	3.79	6.56	8.96	11.13
M2	0.62	0.31	0.17	0.12	0.01	1.99	3.81	7.08	9.70	15.91
M3	12.2	6.38	3.21	2.97	2.13	1.96	3.69	7.28	7.46	10.32
M4	64.9	30.6	15.1	9.56	7.07	1.93	3.70	6.99	10.45	13.77
M5	18.3	9.04	4.61	3.40	**2.46**	1.97	3.92	7.70	10.17	**14.08**
M6	150	110	**70.2**	64.9	51.3	1.94	3.35	**4.07**	10.55	13.38
#Procs.	2	4	8	12	16	2	4	8	12	16

Table 3. Performance of the parallel algorithm using 2, 4, 8, 12, and 16 processors.

tree on a single processor (notice that the number of leaves in the task tree depends on the number of processors). A comparison against the serial algorithm in ILUPACK offers superlinear speed-ups in many cases (see execution times in Table 1) and has little meaning here: the serial and the parallel algorithms compute different preconditioners and to do so, perform different operations. Remarkable speed-ups are attained for Option A/M3 and Option B/M4, M5 and other combinations of option/matrix/number of processors. On the other hand, using 16 processors, a speed-up of only 9.92 is obtained for Option A/M6. The coefficient c_w for this case (see Table 2) is 15%, indicating that all tasks are similar in cost, but the splitting mechanism yields 17 leaf tasks (which concentrate the major part of the computational load) and are to be mapped on 16 processors. As a consequence the distribution of the computational load is unbalanced. A similar case occurs for Option B/M6 on 8 processors. Surprisingly, another similar case, the combination of Option A/M2 on 8 processors, which presents 9 leaves in the task tree, delivers a high speed-up. A closer inspection revealed 8 leaves with very similar costs in the corresponding task tree and a single leaf with of much smaller cost. Due to the dynamic scheduling mechanism, one of the processors receives two tasks, one of them the task of much smaller cost, so that the computational load is not significantly unbalanced.

An analysis of the scalability of the parallel algorithm, though desirable, is difficult. First, in many benchmarks the coefficient matrix is associated with a physical problem, and its dimension (size/sparsity degree) cannot be increased at will (at least easily). Second, even in those cases where the dimension is a parameter that can be adjusted, there may not be a direct relation between the dimension and the cost of the computation of the preconditioner. Finally, in general it is hard to predict the fill-in that will occur during the computation of the preconditioner.

5 Conclusions and Future Work

We have presented a parallel multilevel preconditioner for the iterative solution of sparse linear systems using Krylov subspace methods. The algorithm internally employs the serial routines in ILUPACK. MLND ordering, task splitting, and dynamic scheduling of tasks are used to enhance the degree of parallelism of the computational procedure. Experimental results on a SMM with 16 Itanium2 processors report the performance of our parallel algorithm.

Future work includes:

- To compare and contrast the numerical properties of the preconditioner in ILUPACK and our parallel preconditioner.
- To parallelize the application of the preconditioner to the system.
- To exploit data locality on CC-NUMA architectures, evaluating policies which map tasks to threads taking into consideration the latest tasks assigned to the threads.
- To develop an MPI parallel preconditioner.

References

1. Y. Saad, *Iterative Methods for Sparse Linear Systems*, (SIAM Publications, 2003).
2. OpenMP Arch. Review Board: OpenMP specifications", http://www.openmp.org.
3. M. Bollhoefer, *A robust ILU based on monitoring the growth of the inverse factors*, Linear Algebra Appl., **338**, 201–218, (2001).
4. O. Schenk, M. Bollhöfer and R. A. Römer, *On large scale diagonalization techniques for the Anderson model of localization*, SIAM J. Sci. Comput., **28**, 963–983, (2006).
5. M. Bollhoefer and Y. Saad, *On the relations between ILUs and factored approximate inverses*, SIAM J. Matrix Anal. Appl., **24**, 219–237, (2002).
6. M. Bollhoefer, *A robust and efficient ILU that incorporates the growth of the inverse triangular factors*, SIAM J. Sci. Comput., **25**, 86–103, (2003).
7. M. Korch and Th. Rauber, *A comparison of task pools for dynamic load balancing of irregular algorithms*, Concurrency and Computation: Practice and Experience, **16**, 1–47, (2004).
8. G. Karypis and V. Kumar, *A fast and high quality multilevel scheme for partitioning irregular graphs*, SIAM J. Sci. Comput., **20**, 359–392, (1998).
9. T. Davis, *University of Florida sparse matrix collection*, http://www.cise.ufl.edu-/research/sparse/matrices.
10. T. Davis, *Direct methods for sparse linear systems*, (SIAM Publications, 2006).
11. C. Leiserson and A. Plaat, *Programming parallel applications in Cilk*, SINEWS: SIAM News, (1998).
12. E. Chan, E. S. Quintana-Ortí, G. Quintana-Ortí and R. van de Geijn, *SuperMatrix out-of-order scheduling of matrix operations for SMP and multi-core architectures*, in: Proc. 19th ACM SPAA'07, pp. 116–125, (2007).